13 | 外部函数接口，能不能取代Java本地接口？

2021-12-13 范学雷

《深入剖析Java新特性》 课程介绍 >

讲述：范学雷

时长 08:44 大小 8.01M

你好，我是范学雷。今天，我们一起来讨论 Java 的外部函数接口。

Java 的外部函数接口这个新特性，我写这篇文章的时候，还在孵化期，还没有发布预览版。由于孵化期的特性还不成熟，不同的版本之间的差异可能会很大。我建议你使用最新版本，现在来说就是 JDK 17 来体验孵化期的特性。

Java 的外部函数接口这个特性，有可能会是 Java 自诞生以来最重要的两个特性之一，它和外部内存接口一起，会极大地丰富 Java 语言的生态环境。提前了解一下这样的新特性有助于我们思考现在的技术手段和未来的技术规划。

我们从阅读案例开始，来看一看 Java 的外部函数接口为什么可能会带来这么大的影响，以及它能够给我们的代码带来什么样的变化吧。

阅读案例

我们知道，像 Java 或者 Go 这样的通用编程语言，都需要和其他的编程语言或者环境打交道，比如操作系统或者 C 语言。Java 是通过 Java 本地接口（Java Native Interface, JNI）来支持这样的做法的。 本地接口，拓展了一门编程语言的生存空间和适用范围。有了本地接口，就不用所有的事情都在这门编程语言内部实现了。

比如下面的代码，就是一个使用 Java 本地接口实现的 “Hello, world!"的小例子。其中的 sayHello 这个方法，使用了修饰符 native，这表明它是一个本地的方法。

复制代码

```
public class HelloWorld {
    static {
        System.loadLibrary("helloWorld");
    }

    public static void main(String[] args) {
        new HelloWorld().sayHello();
    }

    private native void sayHello();
}
```

这个本地方法，可以使用 C 语言来实现。然后呢，我们需要生成这个本地方法对应的 C 语言的头文件。

复制代码

```
$ javac -h . HelloWorld.java
```

有了这个自动生成的头文件，我们就知道了 C 语言里这个方法的定义。然后，我们就能够使用 C 语言来实现这个方法了。

复制代码

```
#include "jni.h"
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_HelloWorld_sayHello(JNIEnv *env, jobject jObj) {
    printf("Hello World!\n");
```

```
7 }
```

下一步，我们要把 C 语言的实现编译、链接放到它的动态库里。这时候，就要使用 C 语言的编译器了。

复制代码

```
1 $ gcc -I$(JAVA_HOME)/include -I$(JAVA_HOME)/include/darwin \
2       -dynamiclib HelloWorld.c -o libhelloWorld.dylib
```

完成了这一步，我们就可以运行这个 Hello World 的本地实现了。

复制代码

```
1 java -cp . -Djava.library.path=. HelloWorld
```

你看，一个简单的“Hello, world!"的本地接口实现，需要经历下面这些步骤：

1. 编写 Java 语言的代码（HelloWorld.java）；
2. 编译 Java 语言的代码（HelloWorld.class）；
3. 生成 C 语言的头文件（HelloWorld.h）；
4. 编写 C 语言的代码（HelloWorld.c）；
5. 编译、链接 C 语言的实现（libhelloWorld.dylib）；
6. 运行 Java 命令，获得结果。

其实，在 Java 本地接口的诸多问题中，像代码实现的过程不简洁这样的问题，还属于可以克服的小问题。

Java 本地接口面临的比较大的问题有两个。

一个是 C 语言编译、链接带来的问题，因为 Java 本地接口实现的动态库是平台相关的，所以就没有了 Java 语言“一次编译，到处运行”的跨平台优势；另一个问题是，因为逃脱了 JVM 的语言安全机制，JNI 本质上是不安全的。

Java 的外部函数接口，是 Java 语言的设计者试图解决这些问题的一个探索。

外部函数接口

Java 的外部函数接口是什么样子的呢？下面的代码，就是一个使用 Java 的外部函数接口实现的“Hello, world!"的小例子。我们来一起看看，Java 的外部函数接口是怎么工作的。

复制代码

```
1
2  import java.lang.invoke.MethodType;
3  import jdk.incubator.foreign.*;
4
5  public class HelloWorld {
6      public static void main(String[] args) throws Throwable {
7          try (ResourceScope scope = ResourceScope.newConfinedScope()) {
8              CLinker cLinker = CLinker.getInstance();
9              MemorySegment helloWorld =
10                     CLinker.toCString("Hello, world!\n", scope);
11             MethodHandle cPrintf = cLinker.downcallHandle(
12                     CLinker.systemLookup().lookup("printf").get(),
13                     MethodType.methodType(int.class, MemoryAddress.class),
14                     FunctionDescriptor.of(CLinker.C_INT, CLinker.C_POINTER));
15             cPrintf.invoke(helloWorld.address());
16         }
17     }
18 }
```

在这段代码里，try-with-resource 语句里使用的 ResourceScope 这个类，定义了内存资源的生命周期管理机制。

第 8 行代码里的 CLinker，实现了 C 语言的应用程序二进制接口（Application Binary Interface，ABI）的调用规则。这个接口的对象，可以用来链接 C 语言实现的外部函数。

接下来，也就是第 12 行代码，我们使用 CLinker 的函数标志符（Symbol）查询功能，查找 C 语言定义的函数 printf。在 C 语言里，printf 这个函数的定义就像下面的代码描述的样子。

复制代码

```
1 int printf(const char *restrict format, ...);
```

C 语言里，printf 函数的返回值是整型数据，接收的输入参数是一个可变长参数。如果我们要使用 C 语言打印“Hello, world!”，这个函数调用的形式就像下面的代码。

```
1 printf("Hello World!\n");
```

复制代码

接下来的两行代码（第 13 行和第 14 行代码），就是要把这个调用形式，表达成 Java 语言外部函数接口的形式。这里使用了 JDK 7 引入的 MethodType，以及尚处于孵化期的 FunctionDescriptor。MethodType 定义了后面的 Java 代码必须遵守的调用规则。而 FunctionDescriptor 则描述了外部函数必须符合的规范。

好了，到这里，我们找到了 C 语言定义的函数 printf，规定了 Java 调用代码要遵守的规则，也有了外部函数的规范。调用一个外部函数需要的信息就都齐全了。接下来，我们生成一个 Java 语言的方法句柄（MethodHandle）（第 11 行），并且按照前面定义的 Java 调用规则，使用这个方法句柄（第 15 行），这样我们就能够访问 C 语言的 printf 函数了。

对比阅读案例里使用 JNI 实现的代码，使用外部函数接口的代码，不再需要编写 C 代码。当然，也不再需要编译、链接生成 C 的动态库了。所以，由动态库带来的平台相关的问题，也就不存在了。

提升的安全性

更大的惊喜，来自于外部函数接口在安全性方面的提升。

从根本上说，任何 Java 代码和本地代码之间的交互，都会损害 Java 平台的完整性。链接到预编译的 C 函数，本质上是不可靠的。Java 运行时，无法保证 C 函数的签名和 Java 代码的期望是匹配的。其中一些可能会导致 JVM 崩溃的错误，这在 Java 运行时无法阻止，Java 代码也没有办法捕获。

而使用 JNI 代码的本地代码则尤其危险。这样的代码，甚至可以访问 JDK 的内部，更改不可变数据的数值。允许本地代码绕过 Java 代码的安全机制，破坏了 Java 的安全性赖以存在的边界和假设。所以说，JNI 本质上是不安全的。

遗憾的是，这种破坏 Java 为台完整系的风险，对于应用程序开发人员和最终用户来说，几乎是无法察觉的。因为，随着系统的不断丰富，99% 的代码来自于夹在 JDK 和应用程序之间的第三方、第四方、甚至第五方的类库里。

相比之下，大部分外部函数接口的设计则是安全的。一般来说，使用外部函数接口的代码，不会导致 JVM 的崩溃。也有一部分外部函数接口是不安全的，但是这种不安全性并没有到达 JNI 那样的严重性。可以说，使用外部函数接口的代码，是 Java 代码，因此也受到 Java 安全机制的约束。

JNI 退出的信号

当出现了一个更简单、更安全的方案后，原有的方案很难再有竞争力。外部函数接口正式发布后，JNI 的退出可能也就要提上议程了。

在外部函数接口的提案里，我们可以看到这样的描述：

> JNI 机制是如此危险，以至于我们希望库在安全和不安全操作中都更喜欢纯 Java 的外部函数接口，以便我们可以在默认情况下及时全面禁用 JNI。这与使 Java 平台开箱即用、缺省安全的更广泛的 Java 路线图是一致的。

安全问题往往具有一票否决权，所以，JNI 的退出很可能比我们预期的还要快！

总结

好，到这里，我来做个小结。前面，我们讨论了 Java 的外部函数接口这个尚处于孵化阶段的新特性，对外部函数接口这个新特性有了一个初始的印象。外部内存接口和外部函数接口联系在一起，为我们提供了一个崭新的不同语言之间的协作方案。

如果外部函数接口正式发布出来，我们可能需要考虑切换到外部函数接口，逐步退出传统的、基于 JNI 的解决方案。

这一次学习的主要目的，就是让你对外部函数接口有一个基本的印象。由于外部函数接口尚处于孵化阶段，所以我们不需要学习它的 API。只要知道 Java 有这个发展方向，目前来说就足够了。

如果面试中聊到了 Java 的未来，你不妨聊一聊外部内存接口和外部函数接口，它们要解决的问题，以及能带来的变化。

思考题

其实，今天的这个新特性，也是练习使用 JShell 快速学习新技术的一个好机会。我们在前面的讨论里，分析了下面这段代码。为了方便你阅读，我把这段代码重新拷贝到下面了。

复制代码

```
try (ResourceScope scope = ResourceScope.newConfinedScope()) {
    CLinker cLinker = CLinker.getInstance();
    MemorySegment helloWorld =
            CLinker.toCString("Hello, world!\n", scope);
    MethodHandle cPrintf = cLinker.downcallHandle(
            CLinker.systemLookup().lookup("printf").get(),
            MethodType.methodType(int.class, MemoryAddress.class),
            FunctionDescriptor.of(CLinker.C_INT, CLinker.C_POINTER));
    cPrintf.invoke(helloWorld.address());
}
```

你能不能找一个你熟悉的 C 语言标准函数，试着修改上面的代码，快速地验证一下外部函数接口能不能按照你的预期工作？

需要注意的是，要想使用孵化期的 JDK 技术，需要在 JShell 里导入孵化期的 JDK 模块。就像下面的例子这样。

复制代码

```
$ jshell --add-modules jdk.incubator.foreign -v
|  Welcome to JShell -- Version 17
|  For an introduction type: /help intro

jshell> import jdk.incubator.foreign.*;
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及你的设计和代码。我们下节课见！

注：本文使用的完整的代码可以从GitHub下载，你可以通过修改GitHub上review template代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一

小节的拉取请求代码，请在外部函数接口专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 memory/review/xuelei 的目录下面。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | 外部内存接口：零拷贝的障碍还有多少？

下一篇 14 | 禁止空指针，该怎么避免崩溃的空指针？

精选留言 (8)

写留言

2021-12-13

感知不强，可能还需要时间来沉淀吧

展开

1

ABC

2021-12-15

JShell运行:

jshell --add-modules jdk.incubator.foreign -R--enable-native-access=ALL-UNNAMED

import java.lang.invoke.MethodType;...

展开

共 1 条评论 >

1

ABC

2021-12-14

在JDK18的JShell里面也试了一下,同样报错.

展开

作者回复: JDK 18已经修复了，可能要等到下一个build才能见效。

ABC

2021-12-14

在JShell里面运行,发现报错: java.lang.IllegalCallerException：Illegal native access from: unnamed module @f2a0b8e

在OpenJDK的描述里面提示了，需要设置: java --enable-native-access=M 才能正确运行,但是JShell里面无法设置.老师,请问要怎么才能正常运行?谢谢.

展开

作者回复: 使用“-R--enable-native-access=ALL-UNNAMED”的jshell命令行选项。

共 2 条评论

Jxin

2021-12-14

我的想法是，外部函数接口应该是现有RPC交互的一种暴露接口的新模式。就像暴露http接口，不同语言都暴露相同模式的“外部函数”接口，有一套跨语言的统一规则/协议。调用方与服务方皆遵循统一规则/协议交互。像现在这样，必须调用特定语言的调用规则，在泛化上走不通（要感知异构系统的语言），适用范围就会很局限。

展开

作者回复: 还在孵化器，论证和构造原型为主。现在的设计，已经有点泛化的意思了。试图统一规则和协议的想法，最怕的就是一厢情愿，很难做成。

哦吼掉了

2021-12-14

idea里面的–add-exports页面上一直没加上去，放弃了。jshell17.0.1执行结果，有遇到的小伙伴么？怎么解决的,java.lang.IllegalCallerException：Illegal native access from: unnamed module @6aa8ceb6

| at Reflection.ensureNativeAccess (Reflection.java:112)
| at CLinker.getInstance (CLinker.java:131)

展开

作者回复: 估计Github上的代码没有处理好编译/运行选项，应该设置“--enable-native-access=ALL-UNNAMED”参数。

workspace.xml: <option name="VM_PARAMETERS" value="--enable-preview --enable-native-access=ALL-UNNAMED --add-modules=jdk.incubator.vector,jdk.incubator.foreign" />

共 3 条评论 >

哦吼掉了

2021-12-13

helloworld.c文件中没有写形参，gcc编译会报错。

展开

作者回复: 谢谢，漏掉了。

kimoti

2021-12-13

外部函数借口是不是在内部编译,链接C语言程序,只不过这个过程对Java程序员透明而已？

作者回复: 应该是链接到C的动态库就行。